

EE49 Laboratory 2

UART Communication, Sync, and Channel Correction

Due Date: 5/3/2011 before class

After completing this lab assignment, you must be checked off by a TA in lab and provide your code via email. A few days before the due date, we will send out a sign-up sheet for check-off timeslots. If none of these time slots work for you, it is your responsibility to setup an alternate appointment with the TAs before the due date. Good luck!

Introduction: In Lab 2, our goal is introduce you to aspects of digital signaling, including simple channel codes and the effect of sending these signals over non-ideal communication channels. In Part 1, we will ask you to demodulate a bitstream which is encoded with one widely used digital protocol: Universal Asynchronous Receive/Transmit (UART). This communication protocol is used in nearly every embedded device, from mp3 players to digital watches. It is the basis for many older communication standards like RS-232 and Infrared TV remotes. In Part 2, we will use our UART decoder to receive a 256-color image. First, we will receive the image on an ideal channel. Next, we will receive the image over an LTI non-ideal channel and observe the effects of this non-ideality. Finally, you will design a simple equalizer to estimate the effect of the channel and correct for it before decoding the signal. We begin with a short primer on UART communications.

1 UART Primer

As aforementioned, UART is widely used for digital chip-to-chip communications on embedded systems. While there are many configurations, a common UART packet structure is 10 bits- 1 START bit, 8 DATA bits (one byte), and 1 STOP bit. This structure is shown in figure 1 below, and is commonly referred to as 8n1 (8 data bits, no parity, 1 stop bit). Sometimes an additional bit is included for parity checking (a simple form of error detection), but that is not covered in this lab.

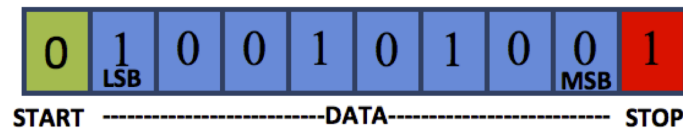


Figure 1: UART Packet Structure

The data byte is sent LSB (Least Significant Bit) first, which means we effectively send the byte "backwards." For instance, if we wanted to send the byte 00101001, we would transmit the following 10-bit sequence: 0100101001 (one start bit (0), the 8 data bits LSB->MSB (10010100), and then one stop bit (1)).

These bits are encoded as voltages: a HIGH voltage (i.e. 5 volts) to represent a 1, and a LOW voltage (i.e. 0 volts) to represent a 0. The START bit is always a LOW state, and the STOP bit is always a HIGH state. Each bit state is held for some period of time - the symbol period - to generate a 'square-wave' type of signal. The amount of time for which we hold a LOW or HIGH bit determines the baud rate (speed) of the link. In most systems, the transmitter and receiver agree upon this baud (or bit-clock) rate in advance. The resulting packet signal is sent along a wire from the transmitter to the receiver.

UART packets can be sent from the transmitter at any point in time, and there can be any amount of time in between packets, hence the **asynchronous** nature of this communication link. In order to achieve synchronization with an incoming packet, the communication wire idles in the HIGH (1) state in between packets. Since the START bit is always a LOW, we know a packet has begun when this transition occurs. After we synchronize to the start of a packet, we use the known baud rate to estimate the center of each data bit, and sample the voltage of the signal at this point. In figure 2 below, we illustrate the start of a packet and the data-bit sampling interval.

After the receiver decodes the entire data packet, we reverse the result (to get the original MSB->LSB) byte, and we're done! We've received one UART packet. The stop bit simply returns the communications wire to the original IDLE (HIGH) state,

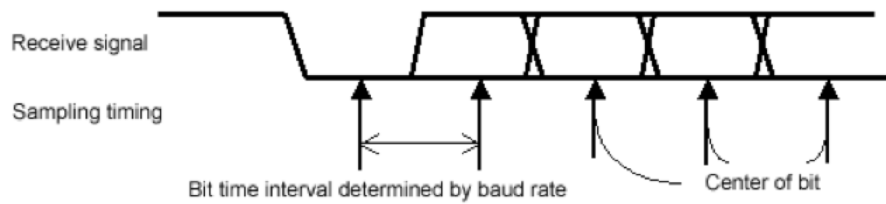


Figure 2: UART Packet Structure

and the receiver begins waiting for the next START bit which signals the beginning of the next packet.

With this primer, you are now ready to start the lab. If you have questions at this point, ask the TAs for clarification on UART, or look for tutorials on the web. There's plenty of information available!

2 The Lab

In lab 1, we asked you to perform signal encoding. In this lab, we ask you to implement the other side of the link: the decoder. Open lab2.vi and take a look at the overarching structure. We have divided this lab into two parts. In Part 1, you will receive a short message from a UART transmitter that we have already built. Open lab2_p2.vi. In Part 2, you will receive an image from a UART encoder in a variety of different ways: First, you will receive the image over an ideal channel. Second, you will receive the image over a non-ideal LTI channel and observe the effects of this non-ideality. Third, you will correct for the LTI channel using a simplified channel equalization (correction) scheme.

3 Part 1: The UART Decoder

Your first task in this lab is to build a functional UART receiver. In Part 1, we have encoded a message (in fact, it is a question) for you using an ASCII encoder and a UART transmitter. The transmitter encodes 0's as 0Volts and 1's as 5Volts. Each character of the message is represented using ASCII encoding. ASCII is a one-byte (8 bit) representation for 256 of the most of the commonly used characters in the English language. As a result, each UART packet you will receive consists of a single

character from this message. The characters of the message are sent in order, so they should be kept in the order that they are received. After decoding the entire stream, you will have received a question. Let's open the `uart_receiver.vi` subVI and get started.

(1) Step 1: Build the Receiver State Machine

In this portion of the lab, we're looking to help you develop a sense of the logic behind a communications receiver. The UART receiver block takes two inputs: an array representing our input signal and the number of samples per UART bit. The latter value is the discrete-time equivalent to the UART baud rate. In this lab, we send 4 samples per UART bit, so the clock rate is 1 bit/4 samples. Another equivalent statement is that the symbol time is 4 samples. We will process the incoming signal sample-by-sample, which is the purpose of the 'for' loop. Our receiver requires a state machine, with two basic states:

- State 1: IDLE, waiting to see a transition from HIGH->LOW. On transition, go to state 2.
- State 2: READ, sample bits in the stream at approximately the middle of each data bit. After we read 8 bits, convert these to a byte and return to IDLE.

We have put in place the basic logical structure you will need to build this state machine. It is your job to make it run!

Build the logic for this state machine, using the nested case structure we provide. Let case 'false' = state IDLE and case 'true' = state READ.

(2) Step 2: Decode the incoming sample stream

With a functional state machine in place, you should be able to decode the incoming UART packets, and load them into an array of 8-bit unsigned integers. Do not forget: each packet is received LSB-first. You'll have to flip the bits before you convert back to decimal!

NOTE: We have provided the `binary_to_decimal` function for your convenience. This function accepts a 8-element array of binary numbers (0 or 1) and converts it to the equivalent decimal value. This decimal value represents one character (in ASCII encoding).

(3) Step 3: Read the question!

After your state machine is working and the incoming data stream is decoded, you should be able to read the question. The conversion from an array of bytes to a string is done for you. If you receive a sensible message, you've written a functional UART decoder! Well done.

4 Part 2: An Image over an LTI Channel, and Dealing with Non-Ideality

The goal for this part of the lab is to introduce you to non-ideal channels and the theory of deconvolution. You will use your UART receiver to decode another stream of data, this time for a 256-color 2d image. Open the file lab2_p2.vi. Use the 'path to bmp' control to point to one of the 256-color images (.bmp files) we provided with the lab directory. There is a bunch of hoopla involved with importing and viewing images in labview. You can ignore all of the picture import and cluster elements aside from the picture data itself. The picture data is comprised (for this image) of a 1D array of bytes, one byte per pixel. Thus, each pixel can take one of 256 colors. The array is ordered by rows from the lowest left pixel to the upper right pixel. Perfect for sending over a UART communication link! This is, in fact, what we'll do: send the picture over the UART channel and view it on the other side.

(1) Verify that your receiver still works

In the upper part of the lab2_p2.vi block diagram, the picture is encoded into a UART stream, sent to your previously designed UART decoder, and then rebuilt into a picture. Without doing anything other than pushing go, this should be able to output the image perfectly! Verify this, and then continue.

(2) Estimating an LTI channel

The second communication link goes over a non-ideal, LTI channel. Plug the output of the LTI channel into a graph node, and take a look at the resulting signal. It is not clear what your receiver will do with this corrupted signal, so your output should be nonsensical at best! At worst, you will not see very much at all. To make matters worse, you do not know what the channel response for this channel is! The taps and gain values are unknown to you. How should you go about decoding this message?

After considering this problem for a moment, notice the additional 'byte' (array element) which is pre-pended to the picture array. It contains decimal 85, which is equivalent to binary 01010101. Your entire first UART packet, including the start and stop bit, will then be 01010101 (remember the data order goes LSB->MSB). We'll call this packet the synchronization & training packet. Now, we give you the following additional pieces of information:

1. Before the start of the first UART packet, the transmitter sends '11111111'. This is simply to ensure that we begin with the communication line IDLEing HIGH.
2. Following these first eight 1's, we send the aforementioned "synchronization" packet.
3. The LTI channel impulse response is no more than 8 taps long ($h[i] = 0 \forall i > 7$), and the channel is noise-free.
4. The **first** tap of the channel impulse is guaranteed to be non-zero ($h[0] \neq 0$)
5. Before the UART transmission starts, the state of the communication line is unknown ($x[i]$ unknown $\forall i < 0$)
6. After this first packet, the communication continues as usual: it sends the pixels of the image.

What this means is that you know the first 18 inputs to the channel. You also observe every output (including the first 18) of the channel. You also know that there are no more than 8 unknown delay tap values. Eighteen known input bits, eighteen measured output bits, and eight unknown tap values. It's up to you to find the right solution to this problem. Don't forget: your inputs are actually over-sampled (4 samples per bit). After our signal passes through the LTI channel, it is then fed to a subVI called 'channel_correction.' This is where you come in. We have provided you with every piece of information you will need to design this block.

Here is your first task: In the 'channel_correction' vi, estimate the delays and tap values of the channel impulse response. *Important note: In deriving a solution, you should use only the sample values after the 8th sample. You should ignore the first 8 sample outputs, since we do not know what went into the channel before the UART transmitter started.**

(3) Correcting for the channel

Your channel_correction vi should actually have two functions - first, estimating the channel response and second, correcting the sample stream once you have come up

with a channel estimate. After you have built the channel estimation block, you will need to design the correction block to remove the effects of this LTI channel for the rest of the sample stream. Without this correction, your UART decoder will not work.

We will consider a simple example to demonstrate how you are going to 'correct' for the channel: Consider the case where you have a channel which has a response of length 2, and we know the first tap is non-zero. The output $y[n]$ at any given time is then

$$y[n] = h[n] * x[n] = \sum_k h[k]x[n-k] = h[0]x[n] + h[1]x[n-1]. \quad (1)$$

Now, let's assume we successfully built a channel estimation block to determine $h[0]$ and $h[1]$. Let's also assume that we know $x[n-1]$, since that was the previous symbol we should have already decoded (think about this for a moment - it's a recursive argument). Since we know everything besides $x[n]$, we just re-arrange the above equation to get an estimate of the current sample:

$$x[n] = \frac{1}{h[0]} (y[n] - h[1]x[n-1]) \quad (2)$$

Work out the math and make sure you understand this idea! Since we know a number of the first samples of the input stream x (the IDLE time and the SYNC packet), you should be able to come up with an algorithm which uses this basic approach.

Your second task: In the 'channel_correction' vi, write the logic for correcting the rest of the sample stream (after the sync packet). Output an "equalized" (corrected) sample array, beginning after the synchronization packet. You should be able to output the same image now, even over a non-ideal channel! You will know its working when the output of this block is a clean looking square signal and your UART receiver correctly decodes the image.

Ask yourself this question: In the presence of random noise, would this channel correction algorithm still work?

Finit. Well done!